RTI COMMON SOFTWARE FRAMEWORK

Michael Hooks, Rich Rybacki, Charles Koplik
TASC, Inc.
55 Walkers Brook Drive
Reading, MA 01867
cmkoplik@tasc.com

KEYWORDS
Run-time Infrastructure (RTI), High Level Architecture (HLA), ORB, middle-ware, Framework, CORBA, IDL, legacy simulation, service repository, exercise management, object management, interest management, ownership management, FOM management

ABSTRACT

The RTI Common Software Framework was developed for STRICOM as part of the HLA Platform Proto Federation project sponsored by DMSO. The Common Software is an object-oriented framework that supports the utilization of the Run-time Infrastructure by Federate simulations. It supports the structured development of RTI support services which are common between multiple federates (or may become common at some future date). The structured environment maximizes reuse potential for federate specific software developed on top of the HLA. The ability to "objectize" the underlying functionality of the RTI empowers the user to customize behavior in specific areas. The granularity of the customized services is arbitrary, dependent on the users or federates needs, and promotes an incremental development process in which small chunks of functionality can be easily modified without impacting the entire system. The Common Software simplifies federate interoperability with the HLA, providing an open object-oriented framework which supports the plug & play of services. Services provide functionality to the federate software for RTI actions such as interest management and time management.

1.0 OVERVIEW

The Platform Protofederation (PPF) is tasked by the Architecture Management Group (AMG) with developing an experimental Federation Execution in conjunction with the High Level simulation Architecture (HLA). As part of this development, we identified a common software approach to provide interoperability with the Run Time Infrastructure (RTI). The common software activity can be considered one of the PPF experiments being conducted under the guidance of US ARMY STRICOM and Defense Modeling and Simulation Office (DMSO). The purpose of the software is to provide a common interface kit for member applications. The HLA Common Software Framework (CSF) allows maximum application interoperability and flexibility while minimizing the cost of development and maintenance due to a common structured architecture that promotes reconfigurability (see Figure 1).

1.1 Common Software Objectives

The Component Service Framework supports the ability to override and extend the simulation behavior through the "plug & play" of services. Services are able to provide functionality to simplify simulation model development and interoperability using the High Level Architecture. The services are able to support CORBA compliant distributed methods, promoting the concept of reconfigurable distributed components. Legacy software systems are connected to the RTI using common services that simplify the interface requirements, aggregate RTI methods, and extend behavior using a flexible and extensible service oriented framework.

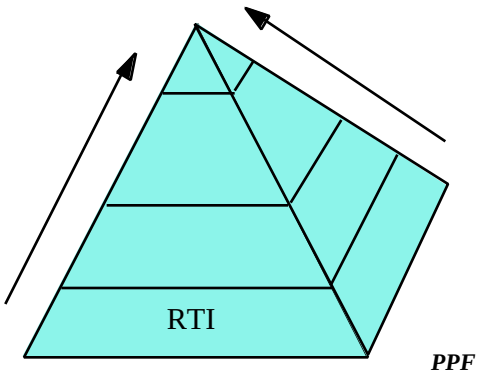


Figure 1 Common Software Provides a Layer that Simplifies the Use of HLA Services

The flexible configuration of services can instigate the evolutionary development of the software used to support the HLA, as new service classes can easily be created to provide additional or different behavior.

1.2 Application to Platform Proto Federation

The Common Software Framework was used to support three diverse Federate members (see Figure 2): BDS-D (a SIMNET tank simulator, BFTT (embedded simulators for carrier, destroyer and weapons) and JTCTS (engineering models for live aircraft and weapons).

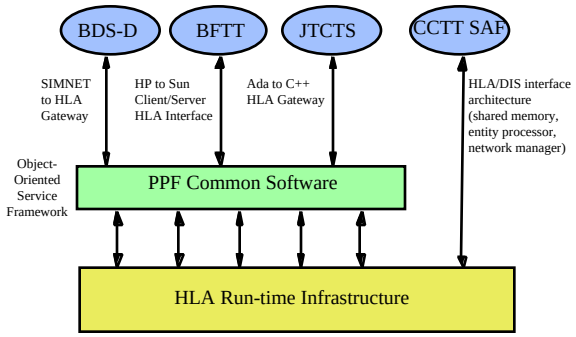


Figure 2 Use of Common Software in Proto Platform Federation Experiment

The integration of the RTI with these legacy systems presented conceptual, as well as implementation difficulties (object orientation, CORBA semantics, process management). The Common Software simplified the software interface and provided additional common functions (e.g., database access to object and attribute information). These service features will be described in the following sections.

2.0 RTI COMMON SOFTWARE

2.1 Simulation Framework

The Common Software Framework is build on top of TASC's Simulation Framework and is really an extension of that framework to the RTI. The Simulation Framework provides a light-weight, CORBA-based communications infrastructure. The Simulation Framework provides a rich set of services for application and GUI components:

- ORB connectivity
- Multi-platform distributed component processing
- Component model encapsulation
- Simulation composition support (build models from object libraries using selection and composition)
- Hierarchical interface typing for maximum software reuse
- Component parameter initialization and data collection support (data probes)
- Database support (object persistence for component library)
- GUI support (multiple GUIs for scenario development, run-time control/analysis)

The TASC Simulation Framework was the first CORBA-based simulation framework developed for DoD (USA TACOM). Figures 3 and 4 show the major component base classes and the various managers that comprise the Framework Manager.

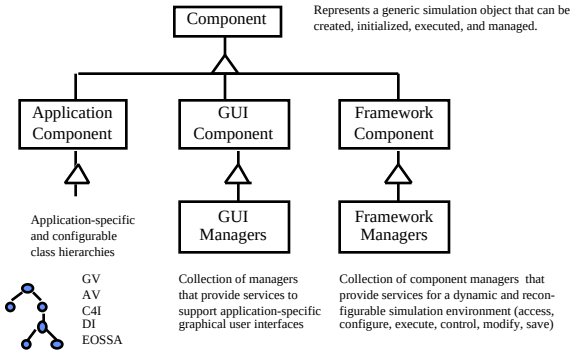
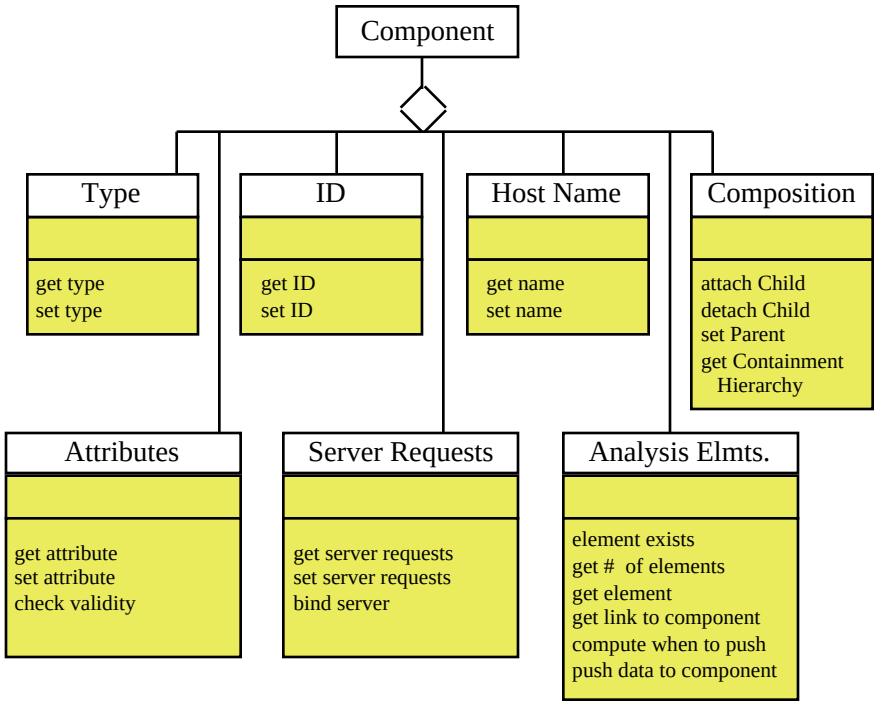


Figure 3 Simulation Framework Component Class Hierarchy

The Component class hierarchy is provided to support common interface and implementation requirements of the components. Framework manager classes provide structure to the component architecture through intelligent agents that provide basic application support (e.g., creation, configuration, component discovery, deletion). The low-level support classes assist the operations of the component and framework classes, reuse of low-level classes simplifies reuse of higher-level classes.

The component class inheritance hierarchy requires any component class to inherit from the existing class hierarchy. The Component class is the ultimate base class, and provides the most basic level of abstraction that can be used for the components. A further specialization of the Component class for application components is provided by the Application Component class. Additional specialization of component classes used for particular application domains can be created through classes that inherit from the Application Component class.

The top of the class hierarchy also branches into the GUI component class used for the GUI (Graphical User Interface) components and the Framework Component used for all framework components. The GUI component and Framework Component classes become further specialized to support manager classes. Manager classes (either GUI or framework) typically represent high level components that aggregate behavior for a set of subservient components. Developers of model components or



Type: classification mechanism used to define the interface capability of the component and the implementation name

ID: identification mechanism for uniquely distinguishing a particular component

Host: location where component exists (computer, cluster)

Attributes: definition and manipulation of model initialization parameters

Composition Manager: specifies notional hierarchy relationship

Analysis Elements: definition and selection of accessible data

Server Requests: definition and access to servers required by component

Figure 4 Simulation Framework Component Class Composition

GUI components can create a class hierarchy that extends this hierarchy as needed for their particular application.

2.2 Common Software Base Class

The base class for the components used in the PPF Experiment inherits from the Application Component. This class will be extended according to the particular needs of the federation member.

The component *create* macros provide the ability for a framework manager to create an instance of the component, and tie the component implementation to the CORBA IDL interface. For the PPF components, the IDL interface that is implemented is named sim_ambassador, as specified by the RTI software. The service declaration macro attaches the service dispatcher to the component class, where the ServiceName is the unique name of the service class.

The component class can contain methods and members specific to the particular federation member requirements and existing designs. The definition of the component class is illustrated by the following example. The service creation macro creates the service dispatcher and instantiates the appropriate service class. The component service becomes registered with the component's service manager and the component service is initialized with information from the component.

The constructor for the component class performs several standard operations that are required by the framework. Only the void constructor is used by the framework creation process, general purpose initialization mechanisms can be used to adjust class parameter values. The first constructor operation, defineName, is used to identify the implementation name of the component. Currently only the name of the component class is required, but advanced services can require run-time typing and identification capabilities (which are supported by the base class). Service creation macros instantiate a service object, connect the object to the base class, and initialize the service.

A component class developed for the HLA can be used to represent different levels of federation object aggregation. A single component class can be used to manage all of the federation objects that are simulated in a particular legacy system. This is the suggested model to be used by the federation members in order to simplify integration and initially concentrate on specific services needed for these types of simulation components. In the future, the component class structure could be used to create a separate component instance for each federated object.

Legacy code is connected to the framework through a component class, specifically in the case of the PPF a C++ class derived from the Common Software base class. The base component class also contains "static extern C" functions that can be accessed with the C and Ada languages.

2.3 Component Services

Extending the application class structure through inheritance can be used to specialize components according to their external or internal service interface requirements. The difficulty with this approach is that it leads to a complicated, multiply inherited, class structure. The Common Service Framework implements a forwarding process for services that can be used to perform component specialization with a flexible mechanism. Services connected to a component class are able to override the behavior of existing methods, as well as extend component behavior with new methods.

Attaching a component service can modify an existing component in several ways; overriding existing behavior of the class definition and adding new behavior that is accessible to the class itself or other application components. Overriding behavior is a fundamental characteristic of Object-Oriented programming languages. The service can also provide new methods that simplify operations that are required by the component class (or legacy code attached to the component). The service methods can also be accessed by other components in the application.

The service attachment mechanism allows the component developer to choose which services are needed based on the component requirements. The developer is able to easily switch between common services to access different behavior. Common services are developed with a similar interface following a structure that promotes reuse. **The services can be viewed as existing in a repository which is accessed during the component configuration process.**

Each component service is implemented as a class which ultimately inherits from the base class Component Service. The Component Service class provides the necessary support to connect the component service to the component through the service dispatch mechanism. The base class also contains a reference to the component that "owns" the service, this reference can be used to invoke methods on the component (see Figure 5).

The common base class for all component services promotes a structured development style which will ensure that common support features can be easily added. Some of the common features that are

implemented in the Component Service class is the reference to the component that "owns" the service, a typing and identification mechanism, and an initialization mechanism.

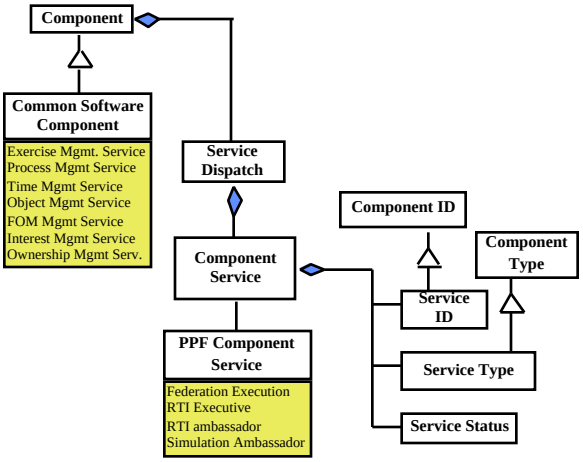


Figure 5 Component Service Class Diagram

3.0 PPF Common Services

A set of RTI common support services was developed specifically tailored to the needs of the Federates participating in the Platform Proto Federation. A subset of these will be delineated in the following sections.

3.1 Exercise Management Services

3.1.1 Summary

The Exercise Management Service provides access to the HLA federation management capabilities. This functionality includes creating, joining, resigning, and destroying federations. The HLA query mechanism is also supported. It is expected that this service class will also support the pause, resume, save, and restore mechanisms when they are implemented within the Run-Time Infrastructure.

This service class will automatically bind to the rti_executive server when the user attempts to either create or join a federation execution. **Prior to starting the rti_executive server, there are several configuration items that can be defined by the user.** These items can use default values if the arguments are not provided to the create or join method invocations. For several of the configuration items, system environment variables can also be used to define behavior.

3.1.2 Class Definition

· Key Class Members

– Federation configuration items.

· Key Class Methods

– createFederation
– Attempt to create a named federation hosted to a particular machine, if the federation already exists the service will proceed by catching the exception.
– joinFederation
– Allow the federate to join an existing federation, the rti_executive and rti_ambassador may be hosted on different machines.
– General Access Methods
– Obtain or set configuration information regarding host names, RTI server name, RID filename.

3.1.3 Key Features

· Configuration Behavior

– Federation Execution easily customized through method interface, default arguments, or environment variables.

· RTI Abstraction

– Simple abstraction to RTI federation management services.
– Hides CORBA server binding operations.
– Supports proper handling of RTI exceptions.

3.1.4 Associations

· rti_executive; create, join, destroy

· rti_ambassador; resign, pause, resume, ...

3.2 FOM Management Service

3.2.1 Summary

The Federation Object Model (FOM) Management Service maintains a database of active classes, attributes, and interactions within the particular federate. Class definitions will contain relationships to the attributes used to describe a FOM class. The FOM classes, attributes, and interactions use a dual identification mechanism consisting of a character string and a unique integer. The FOM Management Service provides access to the mapping between the two representations.

A database of active objects is also maintained by the FOM Management Service. This database will contain the RTI based ID for each object and provide the connection to the FOM class definition associated with the object. The mapping between the RTI object ID and the class definition is currently not fully supported by the RTI.

3.2.2 Class Definition

· Key Class Members

– fomObjectList

FOM class definitions.
–fomInteractionList
FOM interaction definitions.
–objectIdList
Class and object mappings.

· Key Class Methods

– get[ ]ID
Obtain integer identification from string based name.
– get[ ]Name
Obtain string based name from integer identification.
– getObjectIds
Obtain list of active objects matching class name.
getObjectClass
Obtain class name from RTI object ID.

3.2.3 Key Features

· Dynamic Behavior

– The FOM Management Service database is constructed with only the items utilized by the federate simulation.
– Service could be extended to process the RID file, and provide more efficient type checking.

· RTI Name and ID Mapping

– RTI only provides mapping from the string name to the unique integer ID, the FOM Management Service is able to provide the reverse capability.
– Service could be extended to support class hierarchy evaluation (RTI needs to provide class narrowing capability).

· FOM Definition Compile-Time Type Safety

– FOM Management Service has ability to examine compile-time type safety by having the Common Software clients use classes which encapsulate RTI run-time typing mechanisms.

3.2.4 Associations

· tsFomObject

– Support Class; class name, class ID, and list of attributes.

· tsFomInteraction

– Support Class; interaction name, interaction ID, active flag.

· tsFomAttribute

– Support class; attribute name, attribute ID, active flag.

3.3 Object Management Service

3.3.1 Summary

The Object Management Service supports class attribute and interaction publications and transmissions for the High Level Architecture (HLA). Individual objects must be defined by the Federation Object Model (FOM) in terms of the class name and the list of attributes. A client of this service will first state its publication intentions in terms of classes and interactions. Then individual objects can be created and updates to the attributes can be made. Interactions are asynchronous events that can be sent along with a set of parameters at any time during the simulation.

A database of published objects is maintained by the service class in order to simplify client operations. The Object Management Service is able to maintain the latest values of the attributes, only sending values to the RTI that have changed. The database also maintains the RTI update requirements for each object, i.e., the RTI may not request an object, attribute , or interaction to be updated.

3.3.2 Class Definition

· Key Class Members

– proxyList
List of declaration proxy objects.
– publishedInteractionList
List of published interactions.

· Key Class Methods

– publishClass
Define intentions to publish objects belonging to a particular class, where publication results in the update of attribute values.
– publishInteractions
Define intentions to publish interactions.
– createObject
Instantiate an object with a particular set of named attributes (no values).
– updateObject
Provide list of attribute values for RTI update, Object Management Service determines whether the values have changed and are requested by the RTI prior to transmission.

3.3.3 Key Features

· Publication Database

– Maintains previous attribute values sent to RTI, to avoid redundant information being sent.
– Maintains RTI update status on objects.
– Ability to easily extend to persistent database.

· Attribute and Value Support Classes

– Hides CORBA implementation details (e.g., unbounded sequences of 'any's).
– Extensible Object-Oriented design.

· RTI Abstraction

– RTI invocations are implemented with considerations for Platform level simulations, (RTI methods can be aggregated, extended, or modified).

3.3.4 Associations

· FOM Management Service

– String name to ID mapping queries.

· tsDeclarationProxy Support Class

– Hash tables of object attribute and interaction publications.
– List of most recently published attributes.

· tsAttribute, tsValue, tsInteraction Support Classes

– Simplifies RTI integration with CORBA implementation details.
– Encapsulation of RTI transmitted elements into objects.

3.4 Interest Management Service

3.4.1 Summary

The Interest Management Service supports the subscription and reflection mechanisms of the Run-Time Infrastructure (RTI) by enabling the client to express interest in class attributes and interactions. **The subscription process instructs the RTI to only reflect attribute values or send interactions that match the interest declarations.** Currently the RTI only supports class based filtering (i.e., subscription to attributes belonging to a particular class or interactions from a particular class). The interest declarations are forwarded to the federation execution process which is under control of the RTI.

The Interest Management Service is also capable of buffering the reflected attribute values and interaction sent from the other federates. The buffering mechanism will maintain a collection of objects, and their corresponding attribute values, matching the interest criteria. As new values are reflected from the RTI, the old values will be overwritten. Interactions sent to the federate will also be captured in a list. Clients of the Interest Management Service can access the object attribute and interaction parameter data using various query mechanisms.

3.4.2 Class Definition

· Key Class Members

–interestProxyList
List of interest proxy objects.
–interactionInterestList
List of interaction interests.
–objectList
List of received objects.
–interactionList
List of received interactions.

· Key Class Methods

– addClassInterest
Define interest in a particular class, providing the list of named attributes that should be reflected by the RTI.
– addInteractionInterest
Define interest in a particular interaction.
– queryObjectsByName
Obtain collection of objects matching a particular class name, where each object provides access to the list of current attribute values.

3.4.3 Key Features

· Simulation Process Support

– Intercepts RTI attribute and parameter transmissions to collate and package data within a single processing interval.

· Reflected Value and Interaction Database

– Transient database collects all attribute reflections, providing object based query mechanism.
– Queue of interactions.

3.4.4 Associations

· FOM Management Service

– String to ID mapping queries.

· tsInterestProxy Support Class

– Hash tables of attribute and interaction interests.

· tsAttribute, tsValue, tsInteraction Support Classes

– Simplifies RTI integration with CORBA implementation details.
– Encapsulation of RTI transmitted elements into objects.

4.0 SUMMARY

The RTI Common Software Framework supports the application of the RTI to the broad and diverse set of users that is intended for it by the AMG. Since the RTI must be very general (to support all kinds of simulation uses), it has to provide only a base set of primitive capabilities from which a broad variety of applications can then be developed. The Common Software Framework demonstrates how these primitive services can be extended through the use object-oriented middle-ware (much as X-Windows GUI builders provide class libraries that extend the X-Windows primitives). The Common Software facilitates the use of the RTI by the broad community of simulation developers in the following ways:

- Manage Complexity
  - RTI/HLA implementation issues solved in a central location
- Minimize Integration Time
  - Interface can be tailored to specific needs of the platform simulation
  - Results in lower development costs
- Maximize Extensibility
  - Service Repository: allows users to Plug and Play alternative implementations
  - Object-Oriented Methodology allows reusability of services by inheritance -- powerful new services can be created leveraging off of already existing services

This "middle-ware" concept has been successfully tested by three members of the Proto Platform Federation in its recent experiments -- leveraging the benefits of a common solution to several simulation problems (e.g., keeping a database of objects and attributes).

Many of the other HLA Proto Federations also found themselves, by necessity, developing a form of middle-ware for the RTI (although, unlike the PPF, they had not been specifically tasked to investigate this issue) and the STOW '97 program has always assumed the need for an RTI Support Layer. Future work by the community is needed to assess the most appropriate approaches for providing this support layer to the RTI -- as this will be essential in order to achieve the benefits hoped for from a common High Level Architecture for simulation.

6.0 ACKNOWLEDGEMENTS

The authors would like to thank DMSO and STRICOM for supporting this research activiity. We wish to particularly recognize the support of the STRICOM PPF Program Manager, Susan Harkrider, and Lt. Col. Steven Hicks, who suggestions and efforts on behalf of the Platform Proto Federation effort made the project a success. Finally, we wish to acknowledge the contributions of Chris Deschenes and Stephen Bachinsky for their work in supporting the design and development of the Common Softwere.